Behavioral Equivalence

Prof. Clarkson
Fall 2015

Today's music: *Soul Bossa Nova* by Quincy Jones
Review

Previously in 3110:
• Functional programming
• Modular programming
• Interpreters
• Imperative and concurrent programming

Today:
• Reasoning about correctness of programs
Building Reliable Software

• Suppose you work at (or run) a software company.

• Suppose you’ve sunk 30+ person-years into developing the “next big thing”:
  – Boeing Dreamliner2 flight controller
  – Autonomous vehicle control software for Nissan
  – Gene therapy DNA tailoring algorithms
  – Super-efficient green-energy power grid controller

• How do you avoid disasters?
  – Turns out software endangers lives
  – Turns out to be impossible to build software
Approaches to Reliability

• Social
  – Code reviews
  – Extreme/Pair programming

• Methodological
  – Design patterns
  – Test-driven development
  – Version control
  – Bug tracking

• Technological
  – Static analysis
    (“lint” tools, FindBugs, …)
  – Fuzzers

• Mathematical
  – Sound type systems
  – “Formal” verification

Less formal: Techniques may miss problems in programs

All of these methods should be used!

Even the most formal can still have holes:
• did you prove the right thing?
• do your assumptions match reality?

More formal: eliminate with certainty as many problems as possible.
Testing vs. Verification

Testing:
• Cost effective
• Guarantee that program is correct on tested inputs and in tested environments

Verification:
• Expensive
• Guarantee that program is correct on all inputs and in all environments
Edsger W. Dijkstra

Turing Award Winner (1972)

For eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness

"Program testing can at best show the presence of errors but never their absence."

(1930-2002)
Verification

• In the 1970s, scaled to about tens of LOC
• Now, research projects scale to real software:
  – CompCert: verified C compiler
  – seL4: verified microkernel OS
  – Ynot: verified DBMS, web services
• In another 40 years?
Our trajectory

• Proofs about functions
• Proofs about variants
• Proofs about modules

• We're not trying to get all the way to fully machine-checked correctness proofs of large programs
• Rather:
  – help you understand what it means to be correct
  – help you organize your thoughts about correctness of code you write

• Important caveat: no side-effects!
  – specifically, no mutability or I/O
  – exceptions will be fine
Example

```ocaml
let rec even n =
    match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

**Theorem.** For all natural numbers $n$, it holds that $even\ (2*n)$ is true.
Example

(* precondition:  \( n \geq 0 \) *)
(* postcondition:  \( (\text{fact } n) = n! \) *)

let rec fact n =
  if n=0 then 1
  else n * fact (n-1)

Theorem. \( \text{fact} \) is correct—it satisfies its specification.
Example

```ocaml
let rec length = function
| []  -> 0
| _::xs -> 1 + length xs

let rec append xs1 xs2 = match xs1 with
| []  -> xs2
| h::t -> h :: append t xs2
```

**Theorem.** For all lists `xs` and `ys`, it holds that \( \text{length} \ (\text{append} \ xs \ ys) \) is \( \text{length} \ xs + \text{length} \ ys \).
EQUIVALENCE OF EXPRESSIONS
Behavioral equivalence

- Behavioral equivalence: two expressions behave the same
  - always evaluate to same value
Question

Which of these expressions is behaviorally equivalent to 42?

A. `if b then 42 else 42`
   (for an arbitrary Boolean expression b)
B. `let _ = f x in 42`
   (for an arbitrary function f and argument x)
C. `List.hd [42]`
D. All of the above
E. None of the above
Question

Which of these expressions is behaviorally equivalent to 42?

A. `if b then 42 else 42`  
   (for an arbitrary Boolean expression b)
B. `let _ = f x in 42`  
   (for an arbitrary function f and argument x)
C. `List.hd [42]`
D. All of the above
E. None of the above
Behavioral equivalence

- **Behavioral equivalence**: two expressions behave the same
  - always evaluate to same value
  - (or always raise the same exception)
  - (or always *diverge*: don't terminate)

- Write as $e_1 \sim e_2$
  - I would much prefer $e_1 \equiv e_2$, but that symbol isn't available in plain text
Behavioral equivalence

Fundamental axioms about when expressions are behaviorally equivalent:

• **eval**: if $e_1 \rightarrow^* e_2$ then $e_1 \sim e_2$

• **alpha**: if $e_1$ differs from $e_2$ only by consistent renaming of variables then $e_1 \sim e_2$

• **sugar**: if $e_1$ is syntactic sugar for $e_2$ then $e_1 \sim e_2$
Behavioral equivalence

Facts (theorems) about behavioral equivalence:

• **reflexive:** $e \sim e$

• **symmetric:** if $e_1 \sim e_2$ then $e_2 \sim e_1$

• **transitive:** if $e_1 \sim e_2$ and $e_2 \sim e_3$ then $e_1 \sim e_3$

...that is, $\sim$ is an **equivalence relation**
Easy example with ~

let easy x y z = x * (y + z)

Theorem: easy 1 20 30 ~ 50
Proof:
   easy 1 20 30
~ 50               (by eval)
QED
Another easy example

```plaintext
let easy x y z = x * (y + z)
```

Theorem:

for all ints n and m, easy 1 n m ~ n + m

Proof:

easy 1 n m
~ n + m (by eval)
QED

Not so!
Evaluation with unknown values

• That proof wasn't valid according to the small-step semantics:
  – easy 1 n m ->
  – because n and m aren't strictly speaking values
  – they might as well be, though...

• **Symbolic values**: they stand for a value
  – Think of them as "mathematical variables" as opposed to "program variables"
  – They are values; we just don't know what they are
  – We'll allow the semantics to consider them as values

• So we can allow evaluation to continue:
  – easy 1 n m -> x*(y+z){1/x}{n/y}{m/z} -> 1*(n+m)
  – because n+m isn't strictly speaking a value
  – it might as well be, though; guaranteed to produce a value at runtime...
Valuable expressions

• *Valuable*: guaranteed to produce a value
  – No exceptions
  – Always terminates

• If an expression is valuable, then we may use it as though it were already a value in the semantics

• So we can allow evaluation to continue:

  ```
  easy 1 n m
  -> x*(y+z){1/x}{n/y}{m/z}
  -> 1*(n+m)
  -> n+m
  ```
Valuable expressions

Definition of *valuable*:
- a (symbolic) value is valuable
- a variable is valuable
  - at run-time, will be replaced by a value
- any pair, record, or variant built out of valuable expressions is valuable
- an *if* expression is valuable if all its subexpressions are valuable
- a pattern-matching expression is valuable if it is exhaustive
  - non-exhaustive could raise exception at run time
- a function application is valuable if the argument is valuable and the function is *total*: guaranteed to terminate with a value
  - + is total
  - / is *partial*, as is List.hd
Why we need totality

```ocaml
let rec forever x = forever ()
let one x = 1
```

If we didn't require functions to be total, we would conclude

\[
\text{one (forever ())} \rightarrow 1\{\text{forever()} / x\} = 1
\]

hence

\[
\text{one (forever ())) } \sim 1
\]

which violates the definition of behavioral equivalence
Why we need totality

let one x = 1

If we didn't require functions to be total, we would conclude

\[
\text{one (List.hd [])} \\
\rightarrow 1\{\text{List.hd [ ]/x}\} = 1 \\
\]

hence

\[
\text{one (List.hd [])} \sim 1 \\
\]

which violates the definition of behavioral equivalence
Using valuable expressions

```plaintext
let easy x y z = x * (y + z)
```

Theorem: for all ints a, b, and c,
```
   easy a b c ~ easy a c b
```

Proof:
```
   easy a b c
   ~ a * (b + c)   (by eval)
   ~ a * (c + b)   (???)
   ~ easy a c b    (by eval, symm.)
QED
```
Assume basic algebraic properties of the OCaml built-in operators:

- \((r+s)+t \sim r + (s + t)\)
- \(r+s \sim s+r\)
- \(r+0 \sim 0+r \sim r\)
- \(r + (-r) \sim 0\)
- \(r*s \sim s*r\)
- \((r*s)*t \sim r*(s*t)\)
- \(r*0 \sim 0*r \sim r\)
- \(r*1 \sim 1*r \sim r\)
- \(r*(s+t) \sim (r*s)+(r*t)\)
- \((r+s)*t \sim (r*t)+(s*t)\)
- etc.

where \(r, s, t\) must (in general) be valuable
"By math"

Allow use of other mathematical operators that aren't built-in to OCaml:

• Integer exponentiation
• Factorial
• etc.

All arguments must be valuable

e.g.
\[(k+1)! \sim (k+1) \times (k!) \quad \text{(by math)}\]
Using valuable expressions

let easy x y z = x * (y + z)

Theorem: for all ints a, b, and c,
easy a b c ~ easy a c b

Proof:
easy a b c
~ a * (b + c) (by eval)
~ a * (c + b) (by math)
~ easy a c b (by eval, symm.)
QED
Doubles are even

(* requires: n >= 0 *)

let rec even n =
  match n with
  | 0 -> true
  | 1 -> false
  | n -> even (n-2)

Theorem:
for all natural numbers n,
even (2*n) ~ true.
Doubles are even

Theorem:
for all natural numbers n, even (2*n) ~ true.

Proof: by induction. QED

let rec even n =
    match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)

A PL theorist's favorite proof. :)

Doubles are even

Theorem:
for all natural numbers n, even (2*n) \sim true.

Proof: by induction on n

Case: n is 0
Show: even (2*0) \sim true

\[
\text{even (2*0)} \\
\sim \text{true} \quad \text{(eval)}
\]
**Doubles are even**

Theorem:
for all natural numbers \( n \), even \( (2*n) \) ~ true.

Proof: by induction on \( n \)

Case: \( n \) is \( k+1 \), where \( k \geq 0 \)

IH: even \( (2*k) \) ~ true

Show: even \( (2*(k+1)) \) ~ true

even \( (2*(k+1)) \)
~ even \( (2*k+2) \)  (???)
Question

What would justify this proof step?

even \((2 \cdot (k+1)) \sim \text{ even } (2 \cdot k + 2)\)

A. math
B. eval
C. transitivity
D. All the above together
E. None of the above
Question

What would justify this proof step?

\[ \text{even } (2*(k+1)) \sim \text{even } (2*k+2) \]

A. math
B. eval
C. transitivity
D. All the above together
E. None of the above
Congruence

A deep fact about behavioral equivalence:

congruence:
if $e_1 \sim e_2$ then $e\{e_1/x\} \sim e\{e_2/x\}$

aka substitution of equals for equals and Leibniz equality

Congruence is hugely important: enables local reasoning
• replace small part of large program with an equivalent small part
• conclude equivalence of large programs without having to do large proof!
Doubles are even

Theorem: for all natural numbers n, even \( (2 \times n) \) is true.

Proof: by induction on n

Case: n is k+1, where k \( \geq \) 0
IH: even \( (2 \times k) \) is true
Show: even \( (2 \times (k+1)) \) is true

\[
\begin{align*}
\text{even } (2 \times (k+1)) & \\
\sim \text{even } (2 \times k + 2) & \quad \text{(math, congr.)} \\
\sim \text{even } (2 \times k + 2 - 2) & \quad \text{(eval, k} \geq \text{0)} \\
\sim \text{even } (2 \times k) & \quad \text{(math, congr.)} \\
\sim \text{true} & \quad \text{(IH)}
\end{align*}
\]

QED
Review: Induction on natural numbers

Theorem:
for all natural numbers n, P(n).

Proof: by induction on n

Case: n is 0
Show: P(0)

Case: n is k+1
IH: P(k)
Show: P(k+1)

QED
Induction principle

for all properties $P$ of natural numbers,
if $P \ 0$
and (for all $n$, $P \ n$ implies $P \ (n+1)$)
then (for all $n$, $P \ n$)
Upcoming events

- [soon] A5 out; includes design milestone of project which you can start immediately

This is well behaved.

THIS IS 3110
Acknowledgements

This lecture is based on materials from Prof. David Walker (Cornell PhD 2001) et al. at Princeton University from the course COS 326 Functional Programming.

Those materials are in turn based on materials from Prof. Bob Harper (Cornell PhD 1985) et al. at Carnegie Mellon University from the course 15-150 Functional Programming.

Academic genealogy: Constable -> Harper -> Morrisett -> Walker (-> means advised)